Chapter 23

Contax T / Kodax Tri-X

Nunnery — Chichen Itza, Mexico

Three Design Principles

Learning Objectives

- *List the preferred characteristics of an object-oriented application architecture*
- *State the definition of the Liskov Substitution Principle (LSP)*
- *State the definition of Bertrand Meyer's Design by Contract (DbC) programming*
- *Describe the close relationship between the Liskov Substitution Principle and Design by Contract*
- *State the purpose of class invariants*
- *State the purpose of method preconditions and postconditions*
- *Describe the effects weakening and strengthening preconditions have on subclass behavior*
- *Describe the effects weakening and strengthening postconditions have on subclass behavior*
- *State the purpose and use of the Open-Closed Principle (OCP)*
- *State the purpose and use of the Dependency Inversion Principle (DIP)*
- *State the purpose of Code Contracts and how they are used to enforce preconditions, postconditions, and class invariants*

© 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Introduction

Building complex, well-behaved, object-oriented software is a difficult task for several reasons. First, simply programming in C# does not automatically make your application object-oriented. Second, the process by which you become proficient at object-oriented design and programming is characterized by experience. It takes a lot of time to learn the lessons of bad software architecture design and apply those lessons learned to create good object-oriented architectures.

The objective of this chapter is to help you jump-start your object-oriented architectural design efforts. I begin with a discussion of the preferred characteristics of a well-designed object-oriented architecture. I then present and discuss three important object-oriented design principles that you can immediately apply to your software architecture designs to drastically improve performance, reliability, and maintainability.

The three design principles include the Liskov Substitution Principle (LSP), the Open-Closed Principle (OCP), and the Dependency Inversion Principle (DIP). Bertrand Meyer's Design by Contract (DbC) programming is discussed in the context of its close relationship to, and extension of, the Liskov Substitution Principle.

An understanding of these three design principles, coupled with an understanding of how to apply them using the C# programming language, will significantly improve you ability to design robust, object-oriented software architectures.

The Preferred Characteristics Of An Object-Oriented Architecture

From a programmer's perspective, a well-designed, object-oriented architecture manifests itself as an inheritance hierarchy, including a set of abstract data type vertical (inheritance) and horizontal (compositional) relationships, that exhibits several key characteristics. It is 1) easy to understand, 2) easy to reason about, and 3) easy to extend. These characteristics are discussed briefly below.

Easy To Understand: How does this thing work?

A programmer, when shown a component diagram of a complex software system, should be able to understand what it does, or what it is you are trying to do, in about five minutes flat. To do this a software architecture must be designed to be understood.

The organizational complexity of large software systems can be overwhelming if the architecture is poorly designed. An application comprised of even a small number of tightly coupled software components requires significantly more effort to understand than one designed to be understood quickly. An application software architecture must be thoroughly understood by a programmer before the effects of changing its components or adding functionality can be accurately assessed.

Easy To Reason About: What are the effects of change?

The effects of changing pieces of a software application must be fully predictable. Programmers must be confident that the changes they make to one code module will not mysteriously break another, seemingly unrelated, module in the system. If the effects of change can be accurately predicted then the architecture can be reasoned about. The best way to reason about the effects of change is to render code changes unnecessary. (The effects of no change is definitely predictable!)

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Easy To Extend: Where do I add functionality?

Well-designed application architectures accommodate the addition of features and facilitate component reuse. A programmer, when tasked with adding new functionality to an application, must know exactly where to put it. The act of adding functionality should not require the changing of existing code, but rather its extension.

The Liskov Substitution Principle & Design by Contract

Dr. Barbara Liskov and Dr. Bertrand Meyer are both important figures in the object-oriented software research community. The two design principles and guidelines that bear their names are the Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC). These closely related object-oriented design concepts are covered together in this section and can be summarized in the following statement:

> *Subtype objects must be behaviorally substitutable for supertype objects. Programmers must be able to reason correctly about and rely upon the behavior of subtypes using only the supertype behavior specification.*

Reasoning About The Behavior Of Supertypes And Subtypes

Programmers must be able to reason correctly about the behavior of abstract data types and their derived subtypes. The LSP and DbC provide both theoretical and applied foundations upon which programmers can build well-behaved class inheritance hierarchies that facilitate the object-oriented architectural reasoning process.

Relationship Between The LSP And DbC

The LSP and DbC are closely related concepts primarily because they both draw from largely the same body of research in the formulation of their theories. They each address the question of how a programmer should be able to reason about the behavior of a subtype object when it is substituted for a supertype object, they each address the role of method *preconditions* and *postconditions* in the specification of desired object behavior, and they each discuss the role of *class invariants* and how method postconditions should ensure invariant state conditions are preserved. *They both seek to provide a mechanism for programmers to create reliable object-oriented software*. (**Note:** An invariant is a constraint placed upon the state of an object such that upon the completion of an operation on that object the constraint holds true.)

Design by Contract differs from the LSP in its emphasis on the notion of contracts between supertype and subtype. The base class (supertype) is a contractor that may, at runtime, have its interface methods performed by a subcontractor (subtype). Programmers should not need any a priori knowledge of the subtype's existence when they write the code that may come to rely on the subtype's behavior. The subtype, when substituted for the supertype, should fulfill the contract promised by the supertype. In other words, the subtype object should not pull any surprises.

Another difference between the LSP and DbC is that the LSP is more notional, while DbC is more practical. By this I mean no language, as of this writing, directly supports the LSP specifically, with perhaps the exception of the type checking facilities provided by a compiler. Design by Contract, on the other hand, is directly supported by the Eiffel programming language.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

The Common Goal Of The LSP And DbC

The LSB and DbC share a common goal. They both aim to help software developers build correct software from the start. Given this common goal I will occasionally refer to both concepts collectively as the LSP/DbC.

C# Support For The LSP And DbC

With the exception of type checking, C# does not provide direct language support for either the LSP or DbC. However, there are techniques you can use to enforce preconditions and postconditions and to ensure the state of class invariants. Regardless of the level of language support for either the LSP or DbC, programmers can realize significant improvements in their overall class hierarchy designs by simply keeping the LSP and DbC in mind during the design process.

Although the C# language does not directly support the LSP or DbC, as of .NET Framework 4.0, Microsoft added support for *Code Contracts*. I will offer an example of using Code Contracts towards the end of the chapter.

Designing With The LSP/DbC In Mind

The LSP/DbC focuses on the correct specification of supertype and subtype behavioral relationships. By keeping the LSP/DbC in mind when designing class hierarchies programmers are much less likely to create subclasses that implement behavior incompatible with that specified by the base class.

Class Declarations Viewed As Behavior Specifications

A class declaration introduces a new abstract data type into a programmer's environment. The class declaration is, by its very nature, a behavioral specification. The behavior is specified by the set of public interface methods made available to clients, by the set of possible states an object may assume, and by the side effects resulting from method execution.

In C#, class declaration and definition is usually combined. A class that specifies behavior only is known in C# as an interface whereas an abstract class can both specify behavior, and, where necessary, provide behavior implementation.

An abstract data type can adopt the behavioral specification of another abstract data type. (Like one interface extending another interface or a class extending another class.) The former would be the subtype and the latter the supertype. When the supertype is an abstract base class or interface, the subtype inherits only a behavior specification. It must then either implement the specified behavior or further defer the implementation to yet another subtype. When a supertype provides behavior implementation, a subtype may adopt the supertype behavior outright or provide an overriding behavior. It is the correct implementation of this overriding behavior about which the LSP/DbC is most concerned. Programmers can create well-behaved subtypes by employing preconditions, postconditions, and class invariants.

Quick Review

The Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC) programming are closely related principles designed to enable programmers to better reason about subtype behavior.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Preconditions, Postconditions, And Class Invariants

Preconditions, postconditions, and class invariants are the three cornerstones of both the LSP and DbC. I discuss their definitions and application in this section.

Class Invariant

A class invariant is an assertion about an object property that must hold true for all valid states the object can assume. For example, suppose an airplane object has a speed property that can be set to a range of integer values between 0 and 800. This rule should be enforced for all valid states an airplane object can assume. All methods that can be invoked on an airplane object must ensure they do not set the speed property to less than 0 or greater than 800.

Precondition

A precondition is an assertion about some condition that must be true before a method can be expected to perform its operation correctly. For example, suppose an airplane object's speed property can be incremented by some value and there exists in the set of airplane's public interface methods one that increments the speed property anywhere from 1 to 5 depending on the value of the argument supplied to the method. For this method to perform correctly, it must check that the argument is in fact a valid increment value of 1, 2, 3, 4, or 5. If the increment value tests valid then the precondition holds true and the increment method should perform correctly.

The precondition must be true before the method is called, therefore it is the responsibility of the caller to make the precondition true, and the responsibility of the called method to enforce the truth of the precondition.

Postcondition

A postcondition is an assertion that must hold true when a method completes its operations and returns to the caller. For example, the airplane's speed increment method should ensure that the class invariant speed property being 0 <= speed <= 800 holds true when the increment method completes its operations.

An Example

Example 23.1 gives the code for a class named Incrementer. An incrementer object can be incremented by the values 1, 2, 3, 4, or 5 and maintain a state value between 0 and 100. This example illustrates one approach to enforcing method preconditions and postconditions with the help of the Debug.Assert() method. The Debug class is found in the System.Diagnostics namespace.

23.1 Incrementer.cs

```
1   #define DEBUG
2   using System;
3   using System.Diagnostics;
4
5   public class Incrementer {
6     /*******************************************
7       Class invariant: 0 <= Incrementer.val <= 100
8     *******************************************/
9     private int val = 0;
10
11    /*****************************************
12      Constructor Method: Incrementer(int i)
13      precondition: ((0 <= i) &&  (i <= 100))
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
14     postcondition:  0 <= Incrementer.val <= 100
15   ******************************************/
16   public Incrementer(int i){
17     Debug.Assert((0 <= i) && (i <= 100));
18     val = i;
19     Console.WriteLine("Incrementer object created with initial value of: " + val);
20     CheckInvariant();   // enforce class invariant
21   }
22
23   /******************************************
24     Method: void Increment(int i)
25     precondition: 0 < i <= 5
26     postcondition: 0 <= Incrementer.val <= 100
27   ******************************************/
28   public virtual void Increment(int i){
29     Debug.Assert((0 < i) && (i <= 5)); // enforce precondition
30     if((val+i) <= 100){
31       val += i;
32     } else {
33        int temp = val;
34        temp += i;
35        val = (temp - 100);
36       }
37     CheckInvariant();           // enforce class invariant
38     Console.WriteLine("Incremeter value is: " + val);
39   }
40
41   /******************************************
42     Method: void CheckInvariant() - called
43     immediately after any change to class
44     invariant to ensure invariant condition
45     is satisfied.
46   ******************************************/
47   private void CheckInvariant(){
48     Debug.Assert((0 <= val) && (val <= 100));
49   }
50 }// end Incrementer class definition
```

Referring to example 23.1 — First, in order to use the Debug.Assert() method, you'll need to add the #define DEBUG directive at the top of the source file as I've done here.

The Incrementer class has a private instance field of type int named val. The class invariant is specified in comments above the field declaration and indicates that the valid range of values val can assume is 0 through 100. This invariant is enforced with the Debug.Assert() method in the body of the constructor when an instance of Incrementer is created. The invariant is also validated via the CheckInvariant() method.

The Increment() method's preconditions and postconditions are stated in the comment above the method. It indicates that the valid range of increment values the parameter i can assume include 1 through 5, and that when the method completes the class invariant state must be valid. The Increment() method's precondition is checked by the Debug.Assert() method on line 29. The class invariant is checked by calling the CheckInvariant() method on line 37.

Example 23.2 offers a short program that puts the Incrementer class through its paces.

23.2 MainApp.cs

```
1  using System;
2
3  public class MainApp {
4    public static void Main(){
5      Incrementer i1 = new Incrementer(0);
6      i1.Increment(1);
7      i1.Increment(2);
8      i1.Increment(3);
9      i1.Increment(4);
10     i1.Increment(5);
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
11     i1.Increment(6); // throws an assertion exception
12   } // end Main() method
13 }// end MainApp clas definition
```

Referring to example 23.2 — An Incrementer reference named i1 is declared and initialized on line 5. On lines 6 through 11, I call the Increment() method via i1 with different increment values 1, 2, 3, 4, 5, and 6. If you put the #define DEBUG directive at the top of the Incrementer class's source file then the assertion will fail when line 11 executes, causing the Assertion Failed dialog window to display when the program executes. Figure 23-1 shows the results of running this program.

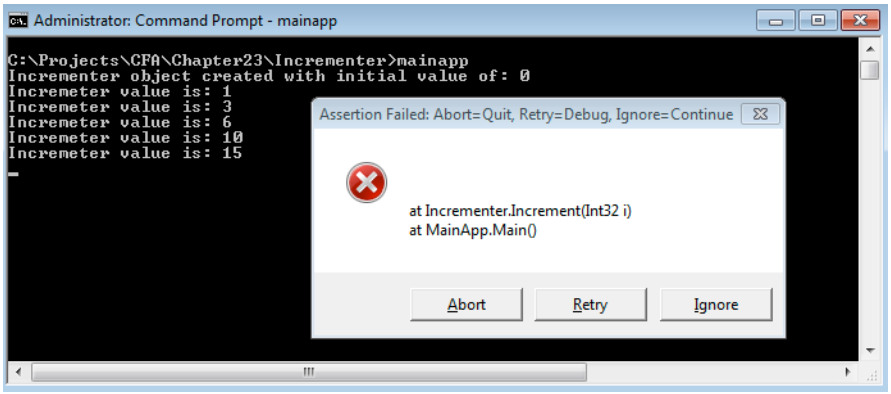

Figure 23-1: Results of Running Example 23.2

Referring to figure 23-1 — You can click the Ignore button to let the program continue execution.

A Note On Using The Debug.Assert() Method To Enforce Pre- and Postconditions

As was just demonstrated, you can place the #define DEBUG directive at the top of a source file to enable the use of the System.Diagnostics.Debug.Assert() method. You can alternatively use the `/d:DEBUG` compiler switch when you compile your code.

The use of the assertion mechanism to enforce method preconditions and postconditions and the state of class invariants is best used during development and testing. Remember, it is the responsibility of the calling program to adhere to a method's documented precondition.

Consider for a moment the MainApp program shown in example 23.2. When a programmer runs this code and gets the error produced by trying to call the Increment method with an invalid precondition, he would then be obliged to fix his code to eliminate the error. From this point forward the assertion mechanism can be safely disabled and the code will run fine.

Using Incrementer As A Base Class

A programmer using the Incrementer class learns from reading its class invariant, precondition, and postcondition specifications how objects of this type can be used in a program and how they should behave. And that is all they should have to know, even when an Incrementer reference points to an object belonging to a class that is derived from Incrementer.

There are several issues that demand the attention of the programmer who plans to extend the functionality of Incrementer. First, he must be aware of the point of view of the client program that will use the derived object. Such code expects certain behavior from Incrementer objects. For example, a client program calling the Increment() method on Incrementer objects can rely on proper behavior *if* the arguments to the method satisfy the precondition of being greater than zero or less than or equal to five. If an object derived from Incrementer is substituted at runtime for an Incrementer object, the derived object must not break the client code by behaving in a manner not anticipated by the client program.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Second, with the expectations of the client code in mind, what rules should a programmer follow when extending the functionality of a base class to ensure the derived object continues to live up to or meet the expectations of the client code? This section explores these issues further.

Example 23.3 gives the code for a class named DerivedIncrementer that extends the functionality of the Incrementer class.

23.3 DerivedIncrementer.cs

```
#define DEBUG
using System;
using System.Diagnostics;

public class DerivedIncrementer : Incrementer {
  /*****************************************
    Class invariant: 0 <= val <= 50
  *****************************************/
  private int val = 0;

  /*****************************************
    Constructor Method: DerivedIncrementer(int i)
    precondition: ((0 <= i) && (i <= 50))
    postcondition: 0 <= val <= 50
  *****************************************/
  public DerivedIncrementer(int i):base(i){
    Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
    val = i;
    Console.WriteLine("DerivedIncrementer object created with value: " + val);
    CheckInvariant();
  }

  /*****************************************
    Method: void Increment(int i)
    precondition: ((0 < i) && (i <= 5))
    postcondition:  0 <= val <= 50
  *****************************************/
  override public void Increment(int i){
    Debug.Assert((0 < i) && (i <= 5)); // enforce precondition
    base.Increment(i);
    if((val+i) <= 50){
      val += i;
    } else {
       int temp = val;
       temp += i;
       val = (temp - 50);
      }
    CheckInvariant();  // check invariant
    Console.WriteLine("DerivedIncrementer value is: " + val);
  }

  private void CheckInvariant(){
    Debug.Assert((0 <= val) && (val <= 50));
  }
} // end DerivedIncrementer class definition
```

Referring to example 23.3 — The DerivedIncrementer class extends Incrementer and overrides its Increment() method. DerivedIncrementer has its own val field which has a different class invariant from that of Incrementer's val. (But this is perfectly OK!) The DerivedIncrementer's version of the Increment() method subscribes to the same precondition as that of the base class version of the method it is overriding, namely, that the values of the integer parameter i can be anything from 0 through 5. Therefore, an object of type DerivedIncrementer will behave the same as objects of type Incrementer. Example 23.4 shows the DerivedIncrementer class in action.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

23.4 MainApp.cs (Mod 1)

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       Incrementer i1 = new Incrementer(0);
6       Incrementer i2 = new DerivedIncrementer(20);
7       i1.Increment(1);
8       i1.Increment(2);
9       i1.Increment(3);
10      i1.Increment(4);
11      i1.Increment(5);
12      Console.WriteLine("----------------------------");
13      i2.Increment(4);
14      i2.Increment(5);
15      i2.Increment(6); // will cause an assertion error
16    } // end Main() method
17  }// end MainApp clas definition
```

Referring to example 23.4 — An Incrementer type reference named i2 is declared on line 6 and initialized to point to an object of type DerivedIncrementer. From lines 13 through 15 the Increment() method is called on the DerivedIncrementer object via i2. When the invalid precondition value of 6 is used in the Increment() method call, the assertion fails as expected. Figure 23-2 shows the results of running this program.

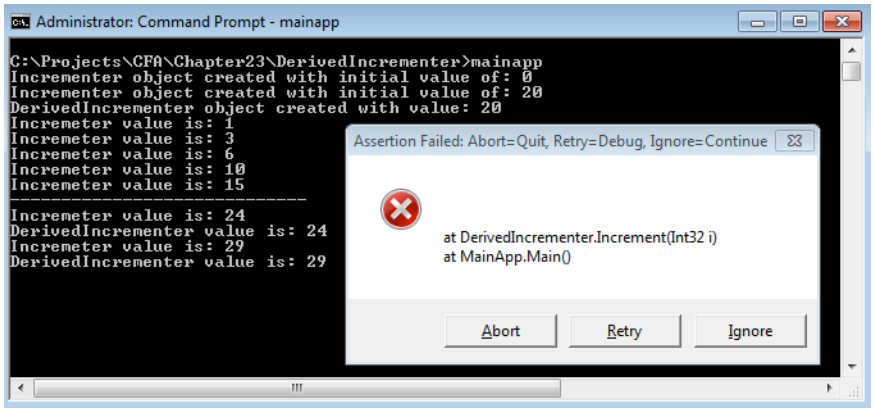

Figure 23-2: Results of Running Example 23.4

Referring to figure 23-2 — The first assertion fails first with the call to the DerivedIncrementer.Increment() method. If you click Ignore you'll get another assertion failure, this time with the base class's Increment() method.

Changing The Preconditions Of Derived Class Methods

The version of the Increment() method in class DerivedIncrementer discussed above implemented the same precondition as the Incrementer class version, namely, that the integer argument passed to the method was in the range 1 through 5. However, it is possible to specify a different precondition for a derived class version of Increment().

In regards to derived class method preconditions, you can go three ways: 1) adopt the same precondition(s), as was illustrated in the previous section, 2) weaken the precondition(s), or 3) strengthen the precondition(s).

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Adopting The Same Preconditions

Derived class methods can adopt the same preconditions as the base class methods they override. The Increment() method in class DerivedIncrementer shown in the previous section adopted the same precondition as the Incrementer class's version of Increment(). When a derived class method adopts the same preconditions as its base class counterpart, its behavior is predictable from the point of view of any client program using a base class reference to a derived class object. In other words, you can safely reason about the behavior of a derived class object whose overriding methods adopt the same preconditions as their base class counterparts.

Weakening Preconditions

Derived class methods can weaken the preconditions specified in the base class methods they override. Weakening can also be thought of as a loosening or relaxing of a specified precondition. The Increment() method in class DerivedIncrementer could have weakened the precondition specified in the base class version of Increment() by allowing a wider range of increment values to be called as arguments. An example of this is shown in the class named WeakenedDerivedIncrementer whose code is given in example 23.5.

23.5 WeakenedDerivedIncrementer.cs

```
#define DEBUG
using System;
using System.Diagnostics;

public class WeakenedDerivedIncrementer : Incrementer {
  /*****************************************
    Class invariant: 0 <= val <= 50
  *****************************************/
  private int val = 0;

  /*********************************************
    Constructor Method: WeakenedDerivedIncrementer(int i)
    precondition: ((0 <= i) && (i <= 50))
    postcondition: 0 <= val <= 50
  *********************************************/
  public WeakenedDerivedIncrementer(int i):base(i){
    Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
    val = i;
    Console.WriteLine("WeakenedDerivedIncrementer object created with value: " + val);
    CheckInvariant();
  }

  /*********************************************
    Method: void Increment(int i)
    precondition: ((0 < i) && (i <= 10))
    postcondition: 0 <= val <= 50
  *********************************************/
  override public void Increment(int i){
    Debug.Assert((0 < i) && (i <= 10)); // enforce precondition
    if((0 <= i) && (i <= 5)){  // remember, it's our job to use the base class correctly!
      base.Increment(i);
    }

    if((val+i) <= 50){
      val += i;
    } else {
       int temp = val;
       temp += i;
       val = (temp - 50);
      }
    CheckInvariant();  // check invariant
    Console.WriteLine("WeakenedDerivedIncrementer value is: " + val);
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
43    }
44
45    /*****************************************
46      Method: void CheckInvariant() - called
47      immediately after any change to class
48      invariant to ensure invariant condition
49      is satisfied.
50    *****************************************/
51    private void CheckInvariant(){
52      Debug.Assert((0 <= val) && (val <= 50));
53    }
54  } // end WeakenedDerivedIncrementer class definition
```

Referring to example 23.5 — The WeakenedDerivedIncrementer class looks a lot like the DerivedIncrementer class with two notable exceptions. First, the precondition on the Increment() method has been relaxed to allow a wider range of increment values. Second, the `if` statement that appears within the body of the Increment() method starting on line 30 ensures the value of i used in the call to the base class version of Increment() obeys its precondition. Example 23.6 shows the WeakenedDerivedIncrementer class being put through its paces in a modified version of the MainApp program.

23.6 MainApp.cs (Mod 2)

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       Incrementer i1 = new Incrementer(0);
6       Incrementer i2 = new DerivedIncrementer(20);
7       Incrementer i3 = new WeakenedDerivedIncrementer(10);
8       i1.Increment(1);
9       i1.Increment(2);
10      i1.Increment(3);
11      i1.Increment(4);
12      i1.Increment(5);
13      Console.WriteLine("----------------------------");
14      i2.Increment(4);
15      i2.Increment(5);
16      Console.WriteLine("----------------------------");
17      i3.Increment(5);
18      i3.Increment(6); // it does not cause an error here...
19      i3.Increment(7); // nor here
20      i3.Increment(8); // nor here
21      i3.Increment(9); // nor here
22      i3.Increment(10); // nor here
23      i3.Increment(11); // ...but here it does!
24    } // end Main() method
25  }// end MainApp clas definition
```

Referring to example 23.6 — A new Incrementer type reference named i3 is declared and initialized to point to an object of type WeakenedDerivedIncrementer. Lines 17 through 23 call the Increment() method via i3. As you can see, WeakenedDerivedIncrementer's version of Increment() allows a wider range of increment values. If steps weren't taken within the body of its Increment() method to obey the contract of Incrementer.Increment() then an assertion error would have occurred on line 18.

However, from the point of view of a programmer who is expecting derived class objects to fulfill the contract of the base class Increment() method, WeakenedDerivedIncrementer objects work just fine because they allow the valid increment ranges of 1 through 5, which is exactly what Incrementer.Increment() methods expect. Figure 23-3 shows the results of running this modified version of MainApp.

Strengthening Preconditions

So far you have seen how a derived class object can be substituted for an Incrementer class object when the derived class's Increment() method adopts the same precondition or weakens the precondition of the Incrementer class's Increment() method. When preconditions are kept the same or weakened in the

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

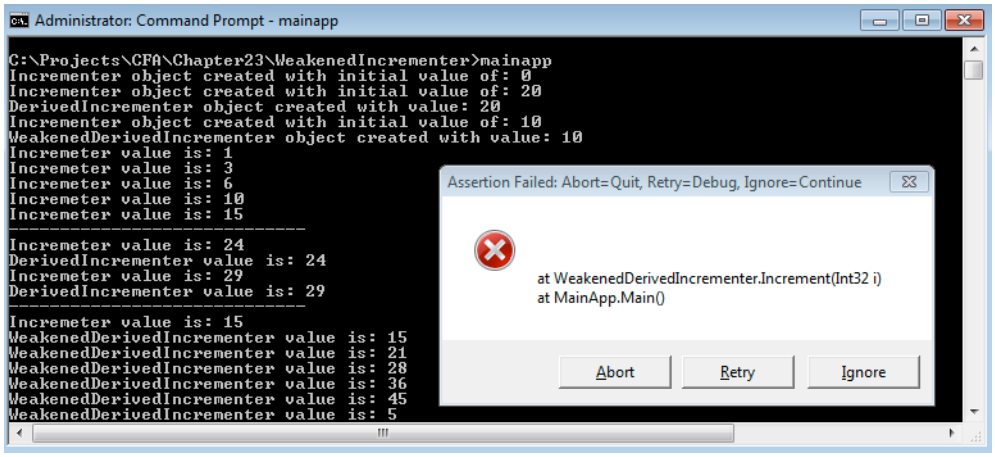
Administrator: Command Prompt - mainapp
C:\Projects\CFA\Chapter23\WeakenedIncrementer>mainapp
Incrementer object created with initial value of: 0
Incrementer object created with initial value of: 20
DerivedIncrementer object created with value: 20
Incrementer object created with initial value of: 10
WeakenedDerivedIncrementer object created with value: 10
Incremeter value is: 1
Incrementer value is: 3
Incremeter value is: 6
Incrementer value is: 10
Incremeter value is: 15
Incremeter value is: 24
DerivedIncrementer value is: 24
Incremeter value is: 29
DerivedIncrementer value is: 29
Incremeter value is: 15
WeakenedDerivedIncrementer value is: 15
WeakenedDerivedIncrementer value is: 21
WeakenedDerivedIncrementer value is: 28
WeakenedDerivedIncrementer value is: 36
WeakenedDerivedIncrementer value is: 45
WeakenedDerivedIncrementer value is: 5
Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue
at WeakenedDerivedIncrementer.Increment(Int32 i)
at MainApp.Main()
Abort
Retry
Ignore

Figure 23-3: Results of Running Example 23.6

overriding methods of a derived class, objects of the derived class type can be substituted for base class objects with little problem. However, if you happen to strengthen the precondition of an overriding derived class method then you will break the code that relies on the original preconditions specified for the base class method.

A strengthening precondition in a derived class method places limits on or restricts the original precondition specified in the base class method it is overriding. In the case of Incrementer and its possible derived classes, the preconditions on an overridden version of Increment() can be strengthened to limit the range of authorized increment values to, say, 1 through 3. This would effectively break any code that relies on the Incrementer's version of the Increment() method that expects the increment values 1 through 5.

Example 23.7 gives the code for a class named StrengthenedDerivedIncrementer whose Increment() method overrides the base class version and strengthens the precondition.

23.7 StrengthenedDerivedIncrementer.cs

```
1   #define DEBUG
2   using System;
3   using System.Diagnostics;
4
5   public class StrengthenedDerivedIncrementer : Incrementer {
6     /*****************************************
7       Class invariant: 0 <= val <= 50
8     *****************************************/
9     private int val = 0;
10
11    /***********************************************
12      Constructor Method: StrengthenedDerivedIncrementer(int i)
13      precondition: ((0 <= i) && (i <= 50))
14      postcondition: 0 <= val <= 50
15    ***********************************************/
16    public StrengthenedDerivedIncrementer(int i):base(i){
17      Debug.Assert((0 <= i) && (i <= 50));  // enforce precondition
18      val = i;
19      Console.WriteLine("StrengthenedDerivedIncrementer object created with value: " + val);
20      CheckInvariant();
21    }
22
23    /*************************************************
24      Method: void Increment(int i)
25      precondition: ((0 < i) && (i <= 3))
26      postcondition: 0 <= val <= 50
27    *************************************************/
28    override public void Increment(int i){
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
29      Debug.Assert((0 < i) && (i <= 3)); // enforce precondition
30      base.Increment(i);
31      if((val+i) <= 50){
32        val += i;
33      } else {
34         int temp = val;
35         temp += i;
36         val = (temp - 50);
37        }
38      CheckInvariant();  // check invariant
39      Console.WriteLine("StrengthenedDerivedIncrementer value is: " + val);
40    }
41
42    /*****************************************
43      Method: void CheckInvariant() - called
44      immediately after any change to class
45      invariant to ensure invariant condition
46      is satisfied.
47    *****************************************/
48    private void CheckInvariant(){
49      Debug.Assert((0 <= val) && (val <= 50));
50    }
51  } // end StrengthenedDerivedIncrementer class definition
```

Referring to example 23.7 — The StrengthenedDerivedIncrementer class places a restriction on the original Increment() method precondition by limiting the authorized increment values to 1 through 3. Example 23.8 shows the StrengthenedDerivedIncrementer class in action. Figure 23-4 shows the results of running this program.

23.8 MainApp.cs (Mod 3)

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       Incrementer i1 = new Incrementer(0);
6       Incrementer i2 = new DerivedIncrementer(20);
7       Incrementer i3 = new WeakenedDerivedIncrementer(10);
8       Incrementer i4 = new StrengthenedDerivedIncrementer(10);
9       i1.Increment(1);
10      i1.Increment(2);
11      i1.Increment(3);
12      i1.Increment(4);
13      i1.Increment(5);
14      Console.WriteLine("-----------------------------");
15      i2.Increment(4);
16      i2.Increment(5);
17      Console.WriteLine("-----------------------------");
18      i3.Increment(5);
19      Console.WriteLine("-----------------------------");
20      i4.Increment(2); // OK so far...
21      i4.Increment(3); // OK here too...
22      i4.Increment(4); // Wait a minute...this should work!
23    } // end Main() method
24  }// end MainApp class definition
```

Changing The Postconditions Of Derived Class Methods

Derived class method postconditions can be adopted, weakened, or strengthened just like preconditions. However, unlike preconditions, where a weakening condition is preferred to a strengthening condition, the opposite is true for postconditions: A derived class method should specify and implement a stronger, rather than weaker, postcondition.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

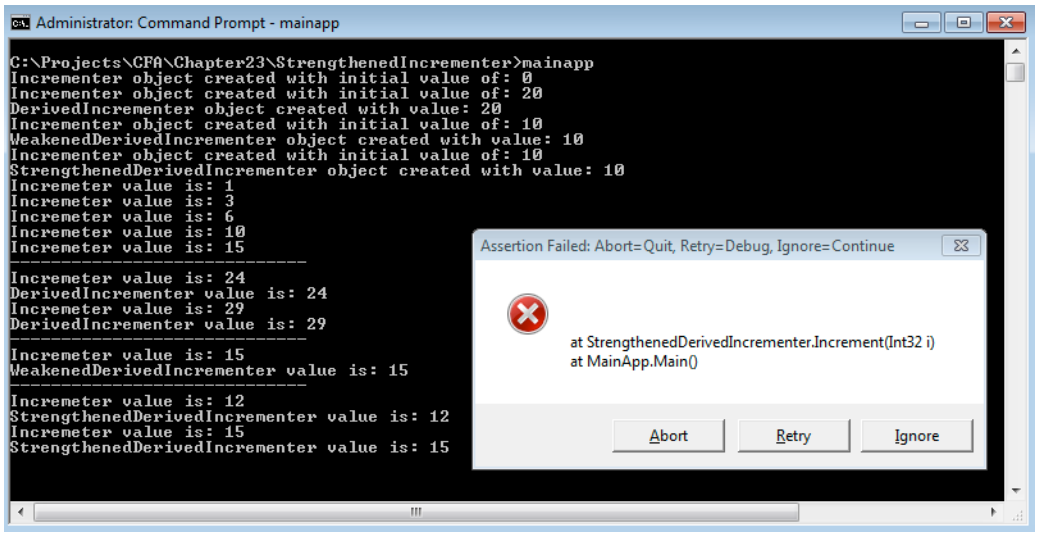
Administrator: Command Prompt - mainapp
C:\Projects\CFA\Chapter23\StrengthenedIncrementer>mainapp
Incrementer object created with initial value of: 0
Incrementer object created with initial value of: 20
DerivedIncrementer object created with value: 20
Incrementer object created with initial value of: 10
WeakenedDerivedIncrementer object created with value: 10
Incrementer object created with initial value of: 10
StrengthenedDerivedIncrementer object created with value: 10
Incremeter value is: 1
Incremeter value is: 3
Incremeter value is: 6
Incremeter value is: 10
Incremeter value is: 15
Incremeter value is: 24
DerivedIncrementer value is: 24
Incremeter value is: 29
DerivedIncrementer value is: 29
Incremeter value is: 15
WeakenedDerivedIncrementer value is: 15
Incremeter value is: 12
StrengthenedDerivedIncrementer value is: 12
Incremeter value is: 15
StrengthenedDerivedIncrementer value is: 15
Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue
at StrengthenedDerivedIncrementer.Increment(Int32 i)
at MainApp.Main()
Abort
Retry
Ignore

Figure 23-4: Results of Running Example 24.8

The Incrementer and its derived class examples shown above each had their own private attribute that was part of each class's invariant. (Incrementer.val, DerivedIncrementer.val, etc.) Each class's Increment() method had a separate postcondition to preserve each class's invariant. The two postconditions did not conflict or contradict and were therefore compatible.

If, on the other hand, Incrementer.val had been declared protected and was inherited and used by its derived classes, then derived versions of the Increment() method would need a postcondition that either maintained the class invariant specified by the Incrementer class (adopting postcondition) or a postcondition that strengthened Incrementer's class invariant (strengthening postcondition).

A weakening postcondition will cause problems. Consider for a moment what would happen if a derived class version of Increment() allowed inherited Incrementer.val to assume values outside the range of those allowed by Incrementer's class invariant specification. Disaster would strike the code sooner than later. (Assuming some code somewhere depended upon Incrementer objects being within their specified, valid states.)

Special Cases Of Preconditions And Postconditions

Method preconditions can specify and enforce more than just the values of method parameters, and postconditions can specify and enforce more than just class invariant states.

A method precondition can, for example, specify that the class invariant must hold true or that a combination of conditions hold true before it can do its job properly. A method postcondition can, in addition to enforcing the class invariant, specify the state of the object or reference the method returns (if any), or it can specify any number of conditions that must hold true upon completion of the method call. The conditions or combination of conditions imposed by derived class overriding method preconditions and postconditions can be weakening or strengthening.

The weakening and strengthening effects of preconditions and postconditions can apply to more than just simple conditions. Method parameter types and return types all play a part and are discussed below.

Method Argument Types

Derived class method preconditions can be weakened or strengthened by their method parameter types. An overriding method must agree with the method it overrides in the *type*, *number*, and *order* of its

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

method parameters. Method parameter types can belong to a type hierarchy. This means that a method parameter might be related to another class via a subtype or supertype relationship.

A derived class method that declares a parameter whose type is a base class to the matching parameter declared by the base class's version of the method is an overriding method. If, however, the derived class method declares a parameter that is a subclass of the parameter type declared by the base class method then the derived class method hides the base class's version of the method. This is due to the transitive nature of subtypes. (*i.e.,* Given two types, Base and Derived, if Derived extends or implements Base, then Derived is a Base but a Base is not a Derived.)

In other words, an overriding method can only provide a weakening precondition with regards to parameter types because to strengthen the parameter type required would result in the declaration of a new method, (i.e., method overloading) (requiring a new type from the point of view of the base class version of the method) not the overriding of the base class method. To illustrate this point assume there exists the class inheritance hierarchy shown in figure 23-5.

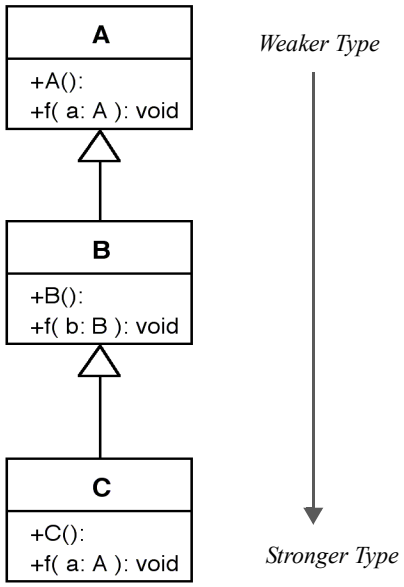

Figure 23-5: Strong vs. Weak Types

Each method f() in each class A and C requires a reference to an object of type A. Method f() in class B specifies a reference to an object of type B. Therefore, method B.f() is an overloading method while method C.f() is an overriding method. Examples 23.9 through 23.11 give the code for classes A, B, and C. Example 23.12 puts these classes through their paces, and figure 23-6 shows the results of running this program.

23.9 A.cs

```
1   using System;
2
3   public class A {
4     public A(){
5       Console.WriteLine("A object created!");
6     }
7
8     public virtual void f(A a){
9       Console.WriteLine("A.f() called!");
10    }
11  }
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

23.10 B.cs

```
1  using System;
2
3  public class B : A {
4    public B(){
5      Console.WriteLine("B object created!");
6    }
7
8    public virtual void f(B b){
9      Console.WriteLine("B.f() called!");
10   }
11 }
```

23.11 C.cs

```
1  using System;
2
3  public class C : B {
4    public C(){
5      Console.WriteLine("C object created!");
6    }
7
8    public override void f(A a){
9      Console.WriteLine("C.f() called!");
10   }
11 }
```

23.12 MainApp.cs

```
1  using System;
2
3  public class MainApp {
4    public static void Main(){
5      A a1 = new A();
6      a1.f(new A()); // A's method called
7
8      Console.WriteLine("------------------");
9
10     A a2 = new B();
11     a2.f(new A()); // A's method called
12     a2.f(new B()); // A's method called
13
14     Console.WriteLine("------------------");
15
16     B b1 = new C();
17     b1.f(new A());  // C's overriding method called
18     b1.f(new B()); // B's overloaded method called
19     b1.f(new C()); // B's overloaded method called
20
21     Console.WriteLine("------------------");
22
23     A a3 = new C();
24     a3.f(new A()); // C's overriding method called
25     a3.f(new B()); // C's overriding method called
26     a3.f(new C()); // C's overriding method called
27   } // end Main() method
28 } // end MainApp program
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

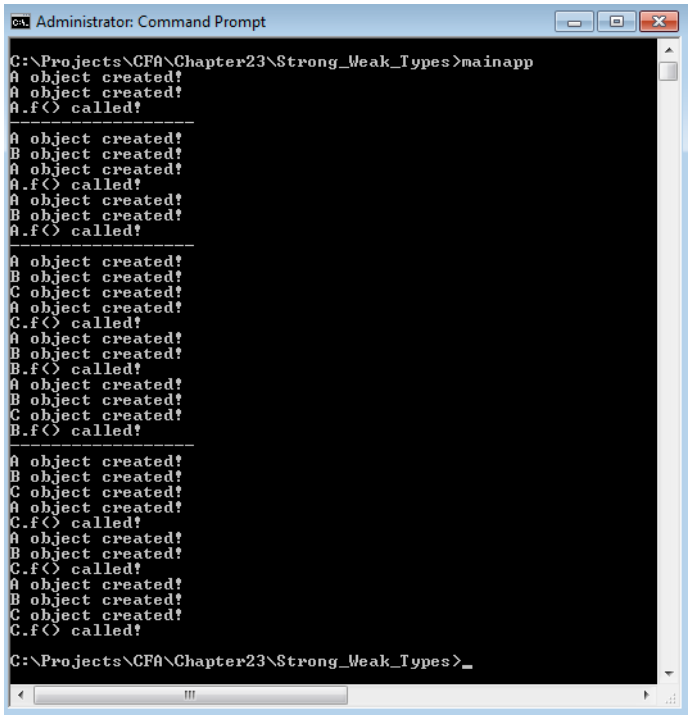

Figure 23-6: Results of Running Example 23.12

Method Return Types

Method return types are considered special cases of postconditions. A reference to an object may be returned from a method as a result of its execution. Refer again to the inheritance hierarchy illustrated in figure 23-5. If a snippet of client code expects a return type from a method to be of a certain type, the method can strengthen that condition and return a subtype of the type expected. This strengthening of return types is in line with the strengthening usually required of postconditions.

Three Rules Of The Substitution Principle

In their book *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Barbara Liskov and John Guttag say that the substitution principle must support three properties: the *signature rule*, the *methods rule*, and the *properties rule*. Each of these rules are discussed below.

Signature Rule

The signature rule deals with the methods published or made public by a type specification. In C# these methods would have public accessibility. For a subtype to obey the signature rule it must support all the methods published by its base class and that each overriding method is compatible with the method it overrides. C# enforces this type compatibility.

Methods Rule

The methods rule says that calls to overriding methods should behave like the base class methods they override. A type may be substitutable from a strictly type perspective but the behavior may be all wrong. Correct behavior of overriding methods is the aim of LSP and DbC.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Properties Rule

The properties rule is concerned with the preservation of provable base class properties by subtype behavior. A subtype should preserve the base class invariant. If a subtype's behavior violates a base class invariant then it is breaking the properties rule.

Quick Review

The preconditions of a derived class method should either adopt the same or weaker preconditions as the base class method it is overriding. A derived class method should never strengthen the preconditions specified in a base class version of the method. Derived class methods that strengthen base class method preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

Method parameter types are considered special cases of preconditions. Preconditions should be weakened in the overriding method, therefore, parameter types should be the same or weaker than the parameter types of the method being overridden. A base class is considered a weaker type than one of its subclasses.

Method return types are considered special cases of postconditions. The return type of an overriding method should be stronger than the type expected by the client code. A subclass is considered a stronger type than its base class.

The Open-Closed Principle

Software systems change over time. Change takes many forms, but changing and evolving system requirements provide the primary catalyst. A software system must accommodate change. It must evolve gracefully throughout its useful life cycle. A software system that is rigid, fragile, and change-resistant exhibits bad design. A software system that is resilient, flexible, and extensible possesses the hallmark characteristics of a well-founded object-oriented architecture. The open-closed principle (OCP) provides the necessary framework for achieving an extensible and accommodating software architecture.

Formulated by Bertrand Meyer, the open-closed principle makes the following assertion:

Software modules must be designed and implemented in a manner that opens them for extension but closes them for modification.

Said another way, changes to software modules should be avoided and new system functionality added by writing new code. It should be noted that writing code that is easy to extend and maintain is a requirement in and of itself. Writing such code takes longer initially but pays a big dividend later. I call it the design dividend.

Achieving The Open-Closed Principle

The key to writing code that conforms to the open-closed principle is to depend upon abstractions, not upon implementations. The reason is because abstractions tend to be more stable. (Correctly designed abstractions are very stable!) This is achieved in C# through the use of abstract base classes or interfaces and dynamic polymorphic behavior. Code should rely only upon the interface methods and behavior promised via abstract methods. A code module that relies only upon abstractions will exhibit the characteristic of being closed to the need for modification yet open to the possibility of extension.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

An OCP Example

A good example of code written with the OCP in mind given in examples 23.13 through 23.24. This code implements a simple naval fleet model where vessels of various types can be constructed with different types of power plants and weapons. Figure 23-7 gives the UML diagram for the naval fleet class inheritance hierarchy. Example 23.24 offers a short program showing the naval fleet classes in action, and figure 23-8 shows the results of running this program.

23.13 Vessel.cs

```
using System;

public abstract class Vessel {
  private PowerPlant _plant;
  private Weapon  _weapon;
  private String  _name;

  // protected properties
  protected Weapon Weapon {
    get { return _weapon; }
  }

  protected PowerPlant Plant {
    get { return _plant; }
  }

  public Vessel(PowerPlant plant, Weapon weapon, String name){
    _weapon = weapon;
    _plant = plant;
    _name = name;
    Console.WriteLine("The vessel " + _name + " created!");
  }

  /* ******************************************************
    Public Abstract Methods - must be implemented in
    derived classes.
  ******************************************************/
  public abstract void LightoffPlant();
  public abstract void ShutdownPlant();
  public abstract void TrainWeapon();
  public abstract void FireWeapon();

  /* ******************************************************
    ToString() Method - may be overridden in subclasses.
  ******************************************************/
  public override String ToString(){
    return "Vessel name: " + _name + " " + _plant + " " + _weapon;
  }
}// end Vessel class definition
```

23.14 PowerPlant.cs

```
using System;

public abstract class PowerPlant {
  private String _model = null;

  public PowerPlant(String model){
    _model = model;
    Console.WriteLine("PowerPlant object created!");
  }

  public abstract void LightoffPlant();
  public abstract void ShutdownPlant();

```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

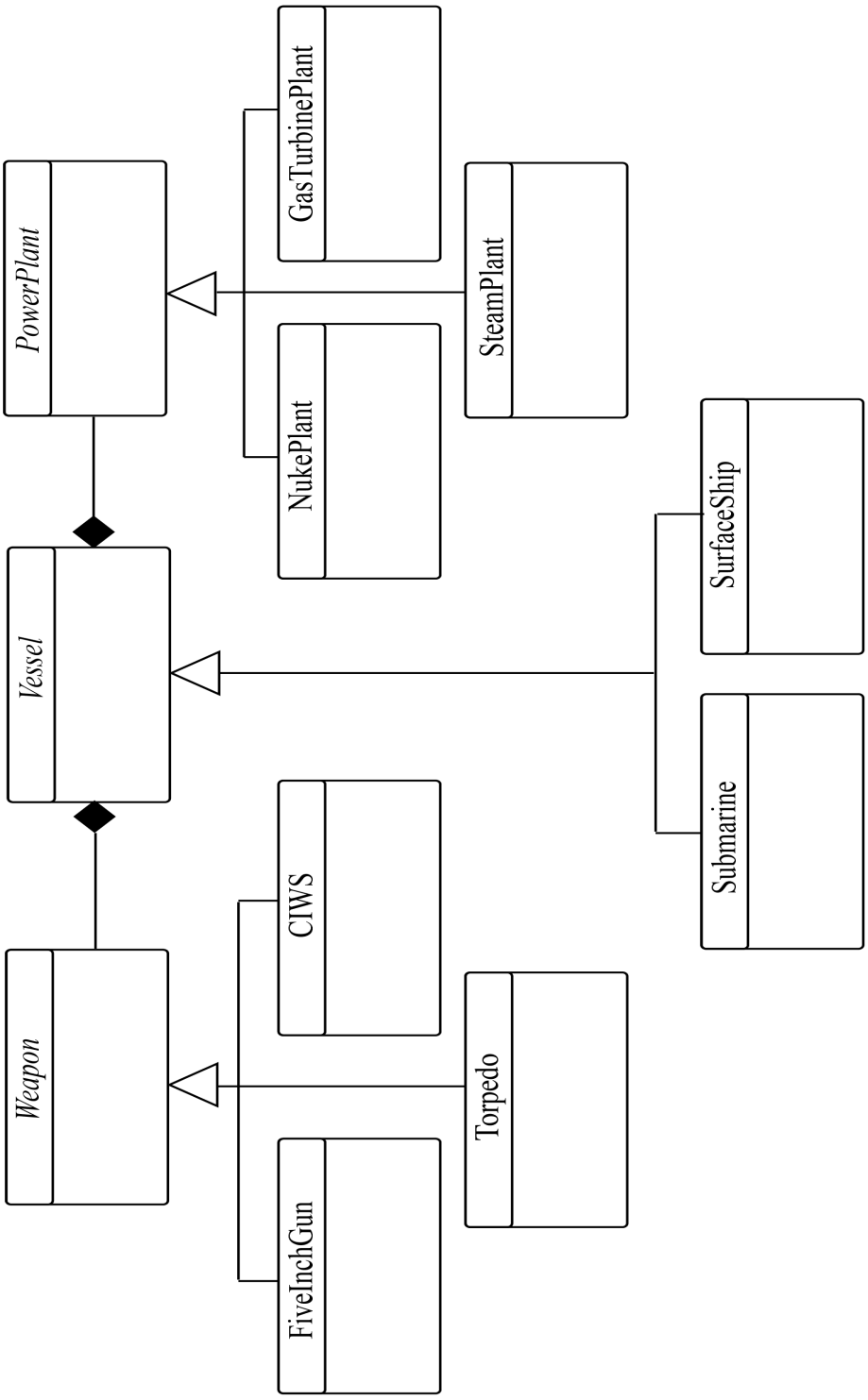

Figure 23-7: Naval Fleet Class Inheritance Hierarchy

© 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
14    public override String ToString(){ return "Power plant model: " + _model; }
15  }
```

23.15 Weapon.cs

```
1   using System;
2
3   public abstract class Weapon {
4     private String _model = null;
5
6     public Weapon(String model){
7       _model = model;
8       Console.WriteLine("Weapon object created!");
9     }
10
11    public abstract void TrainWeapon();
12    public abstract void FireWeapon();
13
14    public override String ToString(){ return "Weapon model: " + _model; }
15  }
```

23.16 CIWS.cs

```
1   using System;
2
3   public class CIWS : Weapon {
4
5     public CIWS(String model):base(model){
6       Console.WriteLine("CIWS object created!");
7     }
8
9     public override void TrainWeapon(){
10      Console.WriteLine("CIWS is locked on target!");
11    }
12
13    public override void FireWeapon(){
14      Console.WriteLine("The CIWS roars to life and fires a zillion bullets at the target!");
15    }
16  }
```

23.17 Torpedo.cs

```
1   using System;
2
3   public class Torpedo : Weapon {
4
5     public Torpedo(String model):base(model) {
6       Console.WriteLine("Torpedo object created!");
7     }
8
9     public override void TrainWeapon(){
10      Console.WriteLine("Torpedo is locked on target!");
11    }
12
13    public override void FireWeapon(){
14      Console.WriteLine("Fish in the water, heading towards target!");
15    }
16  }
```

23.18 FiveInchGun.cs

```
1   using System;
2
3   public class FiveInchGun : Weapon {
4
5     public FiveInchGun(String model):base(model){
6       Console.WriteLine("FiveInchGun object created!");
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

```
7     }
8
9     public override void TrainWeapon(){
10      Console.WriteLine("Five Inch Gun is locked on target!");
11    }
12
13    public override void FireWeapon(){
14      Console.WriteLine("Blam! Blam! Blam!");
15    }
16  }
```

23.19 SteamPlant.cs

```
1   using System;
2
3   public class SteamPlant : PowerPlant {
4
5     public SteamPlant(String model):base(model) {
6       Console.WriteLine("SteamPlant object created!");
7     }
8
9     public override void LightoffPlant(){
10      Console.WriteLine("Steam pressure is rising!");
11    }
12
13    public override void ShutdownPlant(){
14      Console.WriteLine("Steam plant is secure!");
15    }
16  }
```

23.20 NukePlant.cs

```
1   using System;
2
3   public class NukePlant : PowerPlant {
4
5     public NukePlant(String model):base(model) {
6       Console.WriteLine("NukePlant object created!");
7     }
8
9     public override void LightoffPlant(){
10      Console.WriteLine("Nuke plant is critical!");
11    }
12
13    public override void ShutdownPlant(){
14      Console.WriteLine("Nuke plant is secure!");
15    }
16  }
```

23.21 GasTurbinePlant.cs

```
1   using System;
2
3   public class GasTurbinePlant : PowerPlant {
4
5     public GasTurbinePlant(String model):base(model) {
6       Console.WriteLine("GasTurbinePlant object created!");
7     }
8
9     public override void LightoffPlant(){
10      Console.WriteLine("Gas Turbine is running and ready to go!");
11    }
12
13    public override void ShutdownPlant(){
14      Console.WriteLine("Gas Turbine is secure!");
15    }
16  }
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

23.22 Submarine.cs

```
using System;

public class Submarine : Vessel {

  public Submarine(PowerPlant plant, Weapon weapon, String name):base(plant, weapon, name){
    Console.WriteLine("Submarine object created: " + base.ToString());
  }

  public override void LightoffPlant(){
    Plant.LightoffPlant();
  }

  public override void ShutdownPlant(){
    Plant.ShutdownPlant();
  }

  public override void TrainWeapon(){
    Weapon.TrainWeapon();
  }

  public override void FireWeapon(){
    Weapon.FireWeapon();
  }
} // end Submarine class definition
```

23.23 SurfaceShip.cs

```
using System;

public class SurfaceShip : Vessel {

  public SurfaceShip(PowerPlant plant, Weapon weapon, String name):base(plant, weapon, name){
    Console.WriteLine("SurfaceShip object created: " + base.ToString());
  }

  public override void LightoffPlant(){
    Plant.LightoffPlant();
  }

  public override void ShutdownPlant(){
    Plant.ShutdownPlant();
  }

  public override void TrainWeapon(){
    Weapon.TrainWeapon();
  }

  public override void FireWeapon(){
    Weapon.FireWeapon();
  }
} // end SurfaceShip class definition
```

23.24 FleetTestApp.cs

```
using System;

public class FleetTestApp {
  public static void Main(){
    Vessel v1 = new Submarine(new NukePlant("Preasureized Water Mk 85"),
                              new Torpedo("MK 50"), "USS Falls Church");
    v1.LightoffPlant();
    v1.TrainWeapon();
    v1.FireWeapon();
    v1.ShutdownPlant();
  }
}// end FleetTestApp class definition
```

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

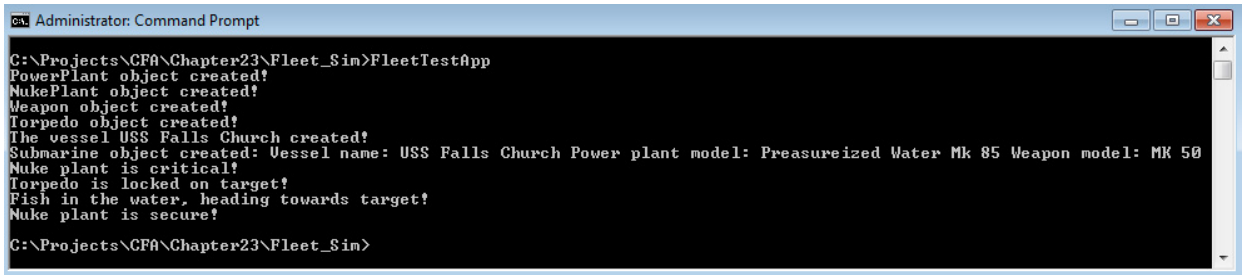
Administrator: Command Prompt
C:\Projects\CFA\Chapter23\Fleet_Sim>FleetTestApp
PowerPlant object created!
NukePlant object created!
Weapon object created!
Torpedo object created!
The vessel USS Falls Church created!
Submarine object created: Vessel name: USS Falls Church Power plant model: Preasureized Water Mk 85 Weapon model: MK 50
Nuke plant is critical!
Torpedo is locked on target!
Fish in the water, heading towards target!
Nuke plant is secure!
C:\Projects\CFA\Chapter23\Fleet_Sim>

Figure 23-8: Results of Running Example 23.22

Quick Review

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending upon software abstractions. In C# this means designing with abstract base classes or interfaces while keeping the goal of dynamic polymorphic behavior in mind. The OCP relies heavily upon the Liskov substitution principle and Design by Contract (LSP/DbC).

The Dependency Inversion Principle

When used together in a disciplined approach, the OCP and the LSP/DbC yield a desirable inversion of program module dependencies that is different from the usual top-down module dependencies attained with functional decomposition. This dependency inversion is generalized into a principle in its own right known as the Dependency Inversion Principle (DIP). Robert C. Martin stated the definition of the DIP in two parts that I've paraphrased here:

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

Characteristics Of Bad Software Architecture

When a software module depends on the details of a lower-level software module it is hard to change and hard to reuse. Consider the software module hierarchy shown in figure 23-9 where high-level modules depend on low-level modules.

Referring to figure 23-9 — The behavior of module A depends on modules B, C, and D. The behavior of module B depends on module E, module C depends on the behavior of modules F and G, and module D depends on module H. A change to module E affects module B, which in turn affects module A. Any inter-module dependencies such as global variables will further complicate the issue. A complex software system sporting this sort of architecture will have the undesirable characteristics of bad design, namely, it will be *fragile*, *rigid*, and *immobile*.

A fragile software architecture is one that breaks in unexpected ways when a change is made to one or more software modules. Fragile software leads to rigid software.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

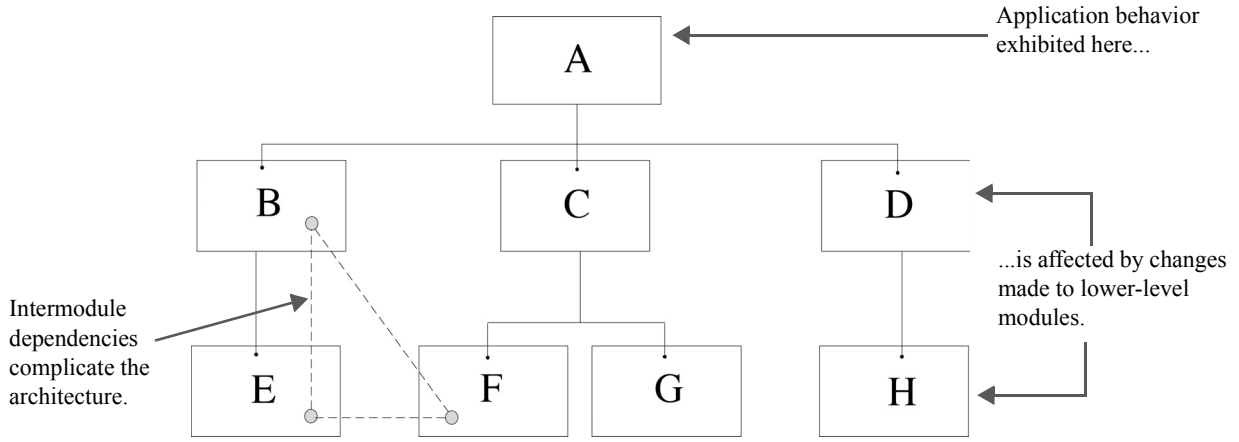

Figure 23-9: Traditional Top-Down Functional Dependencies

A rigid software architecture is one that is so difficult and painful to change that programmers do not want to change it.

An immobile software architecture is characterized by the inability to successfully extract software modules for reuse in other systems. A software module may exhibit desirable behavior but if it is too dependent on other modules or anchored to the application architecture by intermodule dependencies then it will be difficult if not impossible to reuse it in another similar context. If it is easier to rewrite a module from scratch than it is to adopt and reuse the module then the module is immobile.

Characteristics Of Good Software Architecture

Object-oriented software architectures that subscribe to the OCP and the LSP/DbC will depend heavily upon abstractions. These abstractions will appear at or near the top of the software module hierarchy. Refer again to the naval fleet class hierarchy shown in figure 23-7. The Vessel, Weapon, and PowerPlant abstract base classes serve as the foundation for all behavior inherited by the lower-level implementation classes. This inheritance relationship means that the lower-level derived classes are dependent upon the behavior specified by the higher-level base class abstractions.

The key to success with the DIP lies in choosing the right software abstractions. A software architecture based upon the right kinds of abstractions will exhibit the desirable characteristic of being easy to extend. It will be flexible because of its extensibility, it will be non-rigid in that the addition of new functionality via new derived classes will not affect the behavior of existing abstractions. Lastly, software modules that depend upon abstractions can generally be reused in a wider variety of contexts, thus achieving a greater degree of mobility.

Selecting The Right Abstractions Takes Experience

The ability to identify essential software component abstractions takes practice and experience. However, applying the OCP and the LSP/DbC in your object-oriented software architecture design will yield a better design, even if you do not get all the abstractions right the first time around.

Quick Review

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the Dependency Inversion Principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details and that software modules at all hierarchy levels should rely upon

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (*i.e.,* it is flexible and non-rigid). Software modules that conform to the DIP are easier to reuse in other contexts (*i.e.,* they are mobile).

Code Contracts

Microsoft introduced Code Contracts in the .NET Framework version 4.0. Code Contracts allow you to express and enforce preconditions, postconditions, and class/object invariants. Support for Code Contracts is located in the mscorlib.dll library file, but you'll need to download the Code Contracts extensions in order to build assemblies that contain Code Contracts code from the command line. In this section I'll walk you through the Code Contracts extensions download and installation process and give you a short example of Code Contracts in action by modifying the Incrementer class originally listed in example 23.1. Note that this section provides only a very brief introduction to Code Contracts. I recommend you consult the Code Contracts User Manual to dive deeper into the subject.

Downloading And Installing Code Contracts Extensions

Before you can run an application that contains Code Contracts you must run the assembly through a tool called ccrewrite. To obtain this tool you'll need to download the Code Contracts Extensions from Microsoft via the following link:

https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

This link will take you to the Code Contracts for .NET page as is shown in figure 23-10.

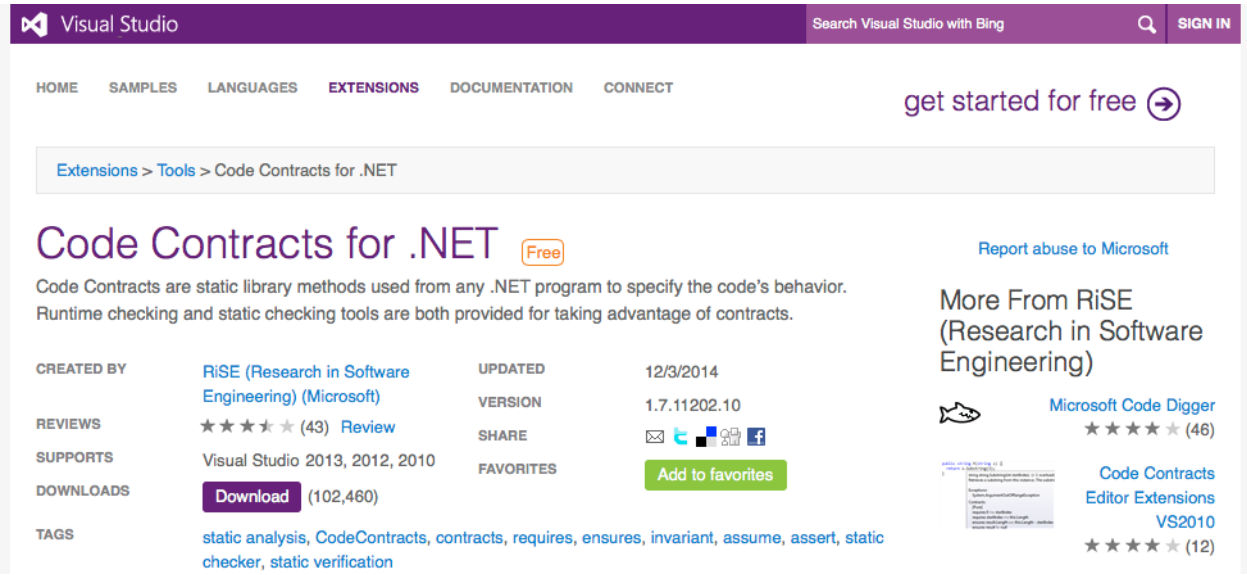

Figure 23-10: Code Contracts for .NET Page

Referring to figure 23-10 — Click the **Download** button and select Run from the pop-up window to install the Code Contracts Extensions. I recommend installing in the default location:

```
C:\Program Files (x86)\Microsoft\Contracts\Bin
```

You'll want to add this path to your PATH environment variable so you can invoke the ccrewrite tool from the command-line.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Code Contracts Example

I think the best way to demonstrate code contracts is to just jump into the deep end of the swimming pool as they say. Example 23.25 lists the Incrementer code rewritten to use Code Contracts to enforce pre-conditions, postconditions, and the class invariant.

23.25 Incrementer.cs (Using Code Contracts)

```
1   using System;
2   using System.Diagnostics.Contracts;
3
4   public class Incrementer {
5     /*********************************************
6       Class invariant: 0 <= Incrementer.val <= 100
7     *********************************************/
8     private int val = 0;
9
10    /*******************************************
11      Constructor Method: Incrementer(int i)
12      precondition: ((0 <= i) &&  (i <= 100))
13      postcondition:  0 <= Incrementer.val <= 100
14    *******************************************/
15    public Incrementer(int i){
16      Contract.Requires<ArgumentException>((0 <= i) && (i <= 100), "i");
17      val = i;
18      Console.WriteLine("Incrementer object created with initial value of: " + val);
19      Contract.Ensures((0 <= val) && (val <= 100));
20    }
21
22    /*********************************************
23      Method: void Increment(int i)
24      precondition: 0 < i <= 5
25      postcondition: 0 <= Incrementer.val <= 100
26    *********************************************/
27    public virtual void Increment(int i){
28      Contract.Requires<ArgumentException>((0 <= i) && (i <= 5), "i");
29      if((val+i) <= 100){
30        val += i;
31      } else {
32          int temp = val;
33          temp += i;
34          val = (temp - 100);
35        }
36      Console.WriteLine("Incremeter value is: " + val);
37      Contract.Ensures((0 <= val) && (val <= 100));
38    }
39
40    /*************************************************
41      Method: void CheckInvariant() - called automatically
42      at the return of each method to ensure invariant
43      condition is satisfied.
44    *************************************************/
45    [ContractInvariantMethod]
46    private void CheckInvariant(){
47      Contract.Invariant((0 <= val) && (val <= 100));
48    }
49  }// end Incrementer class definition
```

Referring to example 23.25 — At first glance there doesn't seem to be a significant difference between this version of Incrementer and the original. The first change you'll note is that I've removed the `#define debug` directive from the top of the file. Next, on line 2, I'm using the System.Diagnostics.Contracts namespace, which contains the Contract class. The first use of the Contract class appears on line 16. Here I'm calling an overloaded version of the Contract.Requires<TException>(Boolean, String) method to

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

enforce a precondition on the parameter i. If the precondition fails, an ArgumentException is thrown. On line 19 I make a call to Contact.Ensures() to enforce the postcondition.

Let's jump down to the CheckInvariant() method. Note now that I have placed the `[ContractInvariantMethod]` attribute above the method definition, and removed the explicit calls to CheckInvariant() from the program. This method will now be called automatically upon each method's return. Also note that I'm making a call to the Contact.Invariant() method within the body of the CheckInvariant() method to enforce the class invariant. One or more calls to the Contract.Invariant() method is the only code allowed within a method tagged with the `[ContractInvariantMethod]` attribute. It is redundant, in this example, to make a call to Contract.Ensures() at the end of each method, since the CheckInvariant() method will be called automatically and will perform the exact same check.

Example 23.26 gives the code for the MainApp class, which remains essentially unchanged from the original example.

23.26 MainApp.cs

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       Incrementer i1 = new Incrementer(0);
6       i1.Increment(1);
7       i1.Increment(2);
8       i1.Increment(3);
9       i1.Increment(4);
10      i1.Increment(5);
11      i1.Increment(6); // will throw an ArgumentException
12    } // end Main() method
13  }// end MainApp clas definition
```

Referring to example 23.26 — The only thing I've changed in this example is the comment on line 11 stating that an ArgumentException will be thrown when the line executes.

OK, to run this program compile both the Incrementer class and the MainApp class together as usual using the following command:

```
csc *.cs
```

This will generate the MainApp.exe assembly. Next, you need to run the MainApp.exe assembly through the ccrewrite tool using the following command:

```
ccrewrite -assemblymode:standard mainapp.exe
```

You should now be able to run the mainapp executable. Figure 23-11 shows the results of compiling, running the ccrewrite tool, and program execution.

Quick Review

Microsoft added Code Contracts beginning with the .NET Framework version 4.0. The Contract class, located in the System.Diagnostics.Contracts namespace, provides support for expressing and enforcing preconditions, postconditions, and class/object invariants. You'll need to download and install the Code Contracts extensions to gain access to the ccrewrite tool. Run the main assembly through ccrewrite before execution.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

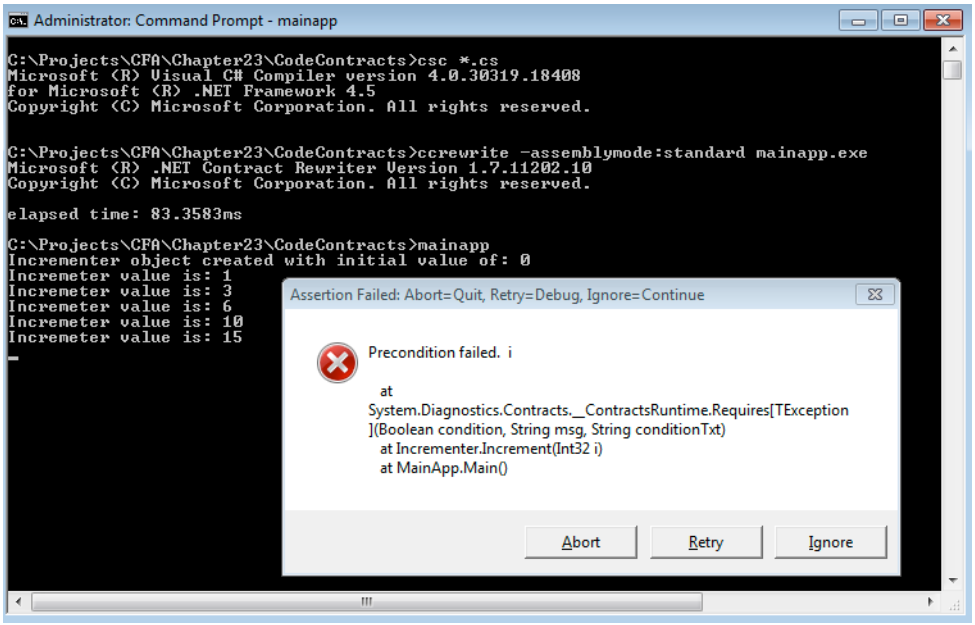
Administrator: Command Prompt - mainapp
C:\Projects\CFA\Chapter23\CodeContracts>csc *.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.18408
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
C:\Projects\CFA\Chapter23\CodeContracts>ccrewrite -assemblymode:standard mainapp.exe
Microsoft (R) .NET Contract Rewriter Version 1.7.11202.10
Copyright (C) Microsoft Corporation. All rights reserved.
elapsed time: 83.3583ms
C:\Projects\CFA\Chapter23\CodeContracts>mainapp
Incrementer object created with initial value of: 0
Incremeter value is: 1
Incremeter value is: 3
Incremeter value is: 6
Incremeter value is: 10
Incremeter value is: 15
Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue
Precondition failed. i
at System.Diagnostics.Contracts.__ContractsRuntime.Requires[TException](Boolean condition, String msg, String conditionTxt)
at Incrementer.Increment(Int32 i)
at MainApp.Main()
Abort
Retry
Ignore

Figure 23-11: Results of Compiling, Calling ccrewrite, and Executing Examples 23.25 and 23.26

Terms and Definitions

The terms and definitions listed in Table 23-1 were used throughout this chapter:

Term	Definition
Abstraction	The separation of the important from the unimportant. (*i.e.*, interface vs. implementation)
Abstract Data Type	A type specification that separates the interface to the type from the type's implementation. An abstract data type represents a set of objects that can be manipulated via a set of interface methods.
Supertype	A data type that serves as a specification for related subtypes.
Subtype	A data type that derives all or part of its specification from another abstract data type. A subtype can inherit the specification of a supertype and then add specialized behavior if required.
Type Specification	A declaration of the behavioral properties of a data type. A specification describes the important characteristics of the data abstraction.
Encapsulation	The act of hiding private implementation details behind a publicly accessible interface.
Precondition	A condition, constraint, or set of constraints that must hold true before a call to an method to ensure its proper operation.

Table 23-1: Terms and Definitions Used in this Chapter

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Term	Definition
Postcondition	A condition, constraint, or set of constraints that must be satisfied when a method completes execution.
Inheritance Hierarchy	A set of type specifications that implement a supertype and subtype relationship.
Class	The declaration of a data type specifying a set of attributes and interface methods common to a set of objects. Methods within the class may implement behavior.
Abstract Class	The declaration of a data type specifying a set of attributes and interface methods common to a set of objects. One or more methods are declared to be abstract and are therefore deferred to subclasses for implementation.
Subclass	A declaration of a data type inheriting all or part of its specification from another, possibly abstract, class. Subtype and subclass are synonymous.
Class/Object Invariant	An assertion about the state of an object which must hold true for all possible states the object may assume.
Code Contracts	Support for the enforcement of preconditions, postconditions, and class/object invariants added to the Microsoft .NET Framework beginning with version 4.0.

Table 23-1: Terms and Definitions Used in this Chapter

Summary

Well-designed software architectures exhibit three characteristics: 1) they are easy to understand, 2) they are easy to reason about, and 3) they are easy to extend.

The Liskov Substitution Principle (LSP) and Bertrand Meyer's Design by Contract (DbC) programming are closely related principles designed to enable programmers to better reason about subtype behavior.

The preconditions of a derived class method should either adopt the same or weaker preconditions as the base class method it is overriding. A derived class method should never strengthen the preconditions specified in a base class version of the method. Derived class methods that strengthen base class method preconditions will render it impossible for programmers to reason about the behavior of subtype objects and lead to broken code should the ill-behaved derived class object be substituted for a base class object.

Method parameter types are considered special cases of preconditions. Preconditions should be weakened in the overriding method, therefore, parameter types should be the same or weaker than the parameter types of the method being overridden. A base class is considered a weaker type than one of its subclasses.

Method return types are considered special cases of postconditions. The return type of an overriding method should be stronger than the type expected by the client code. A subclass is considered a stronger type than its base class.

The open-closed principle (OCP) attempts to optimize object-oriented software architecture design so it can accommodate change. Software modules should be designed so they are closed to modification yet open to extension. The OCP is achieved by depending upon software abstractions. In C# this means designing with abstract base classes or interfaces while keeping the goal of dynamic polymorphic behavior in mind. The OCP relies heavily upon the Liskov substitution principle and Design by Contract (LSP/DbC).

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

The OCP and the LSP/DbC, when applied together, result in the realization of a third design principle known as the Dependency Inversion Principle (DIP). The key to the DIP is that high-level software modules should not rely on low-level details, and that software modules at all hierarchy levels should rely upon abstractions. When a software architecture achieves the goals of the DIP it is easier to extend and maintain (*i.e.,* it is flexible and non-rigid). Software modules that conform to the DIP are easier to reuse in other contexts (*i.e.,* they are mobile).

Microsoft added Code Contracts beginning with the .NET Framework version 4.0. The Contract class, located in the System.Diagnostics.Contracts namespace, provides support for expressing and enforcing preconditions, postconditions, and class/object invariants. You'll need to download and install the Code Contracts extensions to gain access to the ccrewrite tool. Run the main assembly through ccrewrite before execution.

Skill-Building Exercises

1. **Research:** Procure a copy of Bertrand Meyer's excellent book *Object-Oriented Software Construction, Second Edition*, and read it from front to back.

2. **Research:** Procure a copy of Robert C. Martin's book *Designing Object-Oriented C++ Applications Using The Booch Method*. Although the Booch diagramming notation has been superseded by the Unified Modeling Language, and the code examples are given in C++, the C# student will still gain much from reading this excellent work.

3. **Research:** Conduct a web search for the keywords "Liskov substitution principle", "open-closed principle", "dependency inversion principle", and "Meyer design by contract" programming.

4. **API Drill:** Study the System.Diagnostics namespace. List each class and write a brief description about each one.

5. **Code Contracts:** Download and study the Code Contracts User Manual.

6. **API Drill:** Study the System.Diagnostics.Contracts namespace, paying particular attention to the Contracts class and its static methods. List each method and write a brief description of its purpose using your own words.

Suggested Projects

1. **Robot Rat:** Evaluate your latest version of Robot Rat and apply each of the three design principles to your design. What improvements, if any, can be realized by applying each principle? How would your design have to be modified to take full advantage of each of the three design principles?

Self-Test Questions

1. List and describe the preferred characteristics of an object-oriented architecture.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

2. State the definition of the Liskov substitution principle.

3. Define the term “class invariant”.

4. What is the purpose of a method precondition?

5. What is the purpose of a method postcondition?

6. List and describe the three rules of the substitution principle.

7. Write the definition and goals of the open-closed principle.

8. Explain how the open-closed principle uses the Liskov substitution principle and Meyer Design by Contract programming to achieve its goals.

9. Write the definition and goals of the dependency inversion principle.

10. Explain how the dependency inversion principle builds upon the open-closed principle and the Liskov substitution principle/Meyer Design by Contract programming.

References

Barbara Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23,5 (May 1988).

W. Al-Ahmad, *On The Interaction of Programming By Contract and Liskov Substitution Principle*.

Bertrand Meyer, *Applying “Design by Contract”*, IEEE Computer, Vol. 25 Number 10, October 1992, pp. 40 - 51.

Barbara H. Liskov, Jeannette M. Wing, *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November 1994, pp. 1811-1841.

James O. Coplien, *Advanced C++: Programming Styles and Idioms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992. ISBN: 0-201-54855-0

Barbara Liskov, John Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, Massachusetts, 2001. ISBN: 0-201-65768-6

Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN: 0-13-203837-4

Robert C. Martin. *The Dependency Inversion Principle.* http://www.objectmentor.com/resources/articles/dip.pdf

Bertrand Meyer. *Towards practical proofs of class correctness*, to appear in Proc. 3rd International B and Z Users Conference (ZB 2003), Turku (Finland), June 2003, ed. Didier Bert, Springer-Verlag, 2003.

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458. ISBN: 0-13-629155-4

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8, http://www.pulpfreepress.com

Microsoft Website: *Code Contracts for .NET* https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

Microsoft Corporation. *Code Contracts User Manual*, August 14, 2013.

Notes

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved

 © 2015 Rick Miller and Pulp Free Press — All Rights Reserved